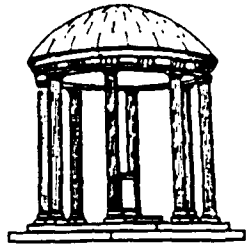
(2)

7 ± 2 Criteria for Assessing and Comparing Spatial Data Structures

TR89-041

October, 1989

Jurg Nievergelt



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

7 ± 2 criteria for assessing and comparing spatial data structures

Jurg Nievergelt

Abstract

Spatial data structures have evolved under the influence of several forces: 1) Database technology, with its emphasis on modeling and logical organization; 2) the long history of data structures developed in response to requirements from other applications; and 3) the recent rapid progress in computational geometry, which has identified typical queries and access patterns to spatial data. Rather than attempting a comprehensive survey of many spatial data structures recently developed, we aim to identify the key issues that have created them, their common characteristics, the requirements they have to meet, and the criteria for assessing how well these requirements are met. As a guideline for tackling these general goals, we begin with a brief history and recall how past requirements from other applications have shaped the development of data structures. Starting from the very early days, five major types of applications generated most of the known data structures. But the requirements of these applications do not include one that is basic to spatial data: That objects are embedded in Euclidian space, and access is most often determined by location in space.

We present six specifically geometric requirements that must be addressed by spatial data structures: Sections 3, 4, 5 discuss the mostly static aspects of how space is organized, how objects are represented and how they are embedded in space; Sections 6, 7, 8 consider the dynamic aspects of how objects are processed. We differentiate three types of processing, of increasing complexity, that call for different solutions: common geometric transformations such as translation and rotation, proximity search, and traversal of the object by different types of algorithms. Together with the general requirement of effective implementability, these seven criteria provide a profile for assessing spatial data structures. This survey leads us to two main conclusions: 1) That the current emphasis on comparative search trees is perhaps unduly influenced by the great success balanced trees enjoyed as a solution to the requirements of earlier applications that rely on single-key access, and 2) that spatial data structures are increasingly of the 'metric' type based on radix partitions of space.

Keywords and phrases

Data management systems, data bases, data structures, geometric and solid modeling, geometric computation, spatial data.

Invited paper presented at the
Symposium on the Design and Implementation of Large Spatial Databases,
University of California at Santa Barbara, July 17-18, 1989.

Affiliation of author: Jurg Nievergelt
Dept.Computer Science, Univ. of North Carolina, Chapel Hill, NC 27514, and
Informatik, ETH, CH-8092 Zurich, Switzerland

Contents

1 The problem

The defining characteristic of spatial data is that it represents objects embedded in Euclidian space, and that access to data is primarily by means of proximity queries of the type 'What objects are in or near a given query region?' Data structures and processing techniques developed in more traditional applications do not adapt readily to spatial data: At worst, they fail to cope with the complexity of the task, and at best, they fail to take advantage of the shortcuts made possible by the rich structure of geometry.

1.1 The conventional data base approach to "non-standard" data

Data base technology developed over the past two decades in response to the needs of commercial data processing, characterized by large, record-oriented, fairly homogeneous data sets, mostly retrieved in response to relatively simple queries: Point queries that ask for the presence or absence of a particular record, interval or range queries that ask for all records whose attribute values lie within given lower and upper bounds. But today, data base research and practice are increasingly concerned with other applications, such as real-time control, hypertext and multimedia. Collectively lumped into the amorphous pool of "non-standard" applications, they confront data base research with new requirements that stretch conventional technology to its limits, and beyond. Among these new applications, none is more important or imposes more stringent new requirements than the management of spatial data, as used in graphics, computer-aided design (CAD), and geographical data bases.

Spatial data lends itself naturally to geometric computation, a topic that has progressed very rapidly in recent years. Thus it is understandable that data base software has yet to take into account the specific requirements and results of geometric computation. Geometric objects are lumped into the amorphous pool of non-standard data, often ignoring any specific properties they might have. But in fact geometric problems possess a great deal of structure to be exploited by efficient algorithms and data structures. This is due primarily to the fact that geometric objects are embedded in space, and are typically accessed through their position in space (as opposed to access by means of identifiers or other non-spatial attributes). Let us begin with a plausible example of how geometry is often introduced as an afterthought to other properties.

1.2 Geometric modeling separated from storage considerations

In this early stage of development of geometric data base technology, we cannot afford to focus on modeling to the exclusion of implementation aspects. In interactive graphics and CAD *the real issue is efficiency*: 1/10-th of a second is the limit of human time resolution, and a designer works at maximal efficiency when "trivial" requests are displayed "instantaneously". This allows a couple of disk accesses only, which means that geometric and other spatial attributes must be part of the retrieval mechanism if common geometric queries (intersection, inclusion, p[illegible] queries) are to be handled efficiently.

Data base terminology makes a sharp distinction betw[illegible] the *logical view* presented to the user and the *physical aspects* seen by the implementor. It has been possible to uphold this distinction in conventional data base applications because data structures that allow efficient handling of records identified by a primary key are so well understood that they can be hidden from the user. The same distinction is premature for geometric data bases: In interactive applications such as CAD, efficiency is today's bottleneck, and until we understand geometric storage techniques better we may not be able to afford the luxury of studying geometric modeling divorced from physical storage. Consider the familiar example of a spatial object represented (or approximated) as the union of a set of disjoint tetrahedra. If the latter are stored in a relational data base by using the *boundary representation (BR)* approach, a tetrahedron t is given by its faces, a face f by its bounding edges, an edge e by its endpoints p and q. Four relations *tetrahedra*, *faces*, *edges* and *points* might have the following tuple structure:

- *tetrahedra* : a pair (t_i, f_k) of identifiers for a tetrahedron t_i and one of its faces f_k.
- *faces* : a pair (f_k, e_j) of identifiers for a face f_k and one of its edges e_j.
- *edges* : a triple (e_j, p_m, p_n) of identifiers for an edge e_j and its two points p_m and p_n.
- *points* : a triple (p_n, x, y, z): p_n is the identifier of a point, x, y, z its coordinates.

This representation smashes as simple an object as a tetrahedron into parts spread over different relations and therefore over the storage medium. The question whether a tetrahedron t intersects a given line L is answered by intersecting each of its faces f_k with L. If the tuple (t_i, f_k) in the relation *tetrahedra* contains the equation of the corresponding plane, the intersection point of the plane and the line L can be computed without accessing other relations. But determining whether this intersection point lies inside or outside the face f_k requires accessing tuples of *edges* and *points*, i.e. accessing different blocks of storage, resulting in many more disk accesses than the geometric problem requires.

Efficient disk access requires, at least, retrieving as a unit all the data that defines a basic volume element such as a tetrahedron. But we can aim for more and use geometric properties to design representations that answer certain proximity queries more efficiently. For example, by representing a primitive object by a suitably chosen set of parameters. A tetrahedron can be defined by 12 parameters in many useful ways: 4 vertices of 3 coordinates each, or 4 faces, each of whose equations has 3 coefficients; or as the minimal containing sphere (4 parameters) and 4 vertices that lie on the surface of the sphere, each one given by 2 spherical coordinates. The 4–parameter sphere serves as a simple container to provide a negative answer to many intersection queries more efficiently than the 12–parameter tetrahedron can. If we represent a more complex polyhedron as a union of tetrahedra, the latter can often be ordered so as to support efficient processing (for example through hierarchical decomposition, which is particularly effective for convex solids). The point of these examples is that, if we forget that geometric objects are embedded in space and treat them like any other data, we fail to take advantage of the rich structure of geometry, with grievous consequences for efficiency.

1.3 Structures for spatial data bases

In recognition of the fact that modeling is only the peripheral part of spatial data management, and physical storage is the key to efficiency, many data structures, old and new, have been studied in conjunction with computational geometry and CAD: [EL 80] is a comprehensive bibliography, [Me 84], [Gun 88], and [Sa 89] are survey books. The majority of the data structures proposed for geometric computation are variations and combinations of a few themes: Mostly, multidimensional trees based on comparative search, used to organize different types of objects such as points, intervals, segments, or rectangles. A few structures, such as quad trees and oct-trees, are based on hierarchical radix partitions of space.

Recent years have seen intense efforts in studying spatial data structures, and many new have been developed (for example [Gut 84], [SRF 87], [SK 88], [HSW 89]). Rather than attempting a comprehensive survey of many spatial data structures recently developed, I aim to identify the key issues that have created them, their common characteristics, the requirements they have to meet, and the criteria for assessing how well these requirements are met. As a guideline for tackling this more general endeavor, let us begin with a brief history and recall how past requirements from other applications have shaped the development of data structures. Starting from the very early days, the following sections summarize five types of applications whose specific requirements led to most of the known data structures. This survey leads us to two main conclusions: 1) That the current emphasis on comparative search trees is perhaps unduly influenced by the great success balanced trees enjoyed as a solution to the requirements of applications that rely on single-key access, and 2) that spatial data structures are increasingly of the 'metric' type based on radix partitions of space.

2 The development of data structures: Requirements drive design

This section sketches the evolution of data structures. We observe that each new era has created new techniques to address concerns not adequately met by prior developments. Managing spatial data brings new problems, characterized by highly structured geometric objects that are accessed primarily on the basis of their location in space and relation to other objects, rather than their inherent properties as objects.

2.1 Scientific computation: Static data sets

Numerical computation in science and engineering mostly leads to linear algebra and hence matrix computations. Matrices are static data sets: The values change, but the shape and size of a matrix rarely does - this is true even for most sparse matrices, such as band matrices, where the propagation of nonzero elements is bounded. Arrays were Goldstine and von Neumann's answer to the requirement of random access, as described in their venerable 1947 report "Planning and coding of problems for an electronic computing instrument". FORTRAN '54 supported arrays and sequential files, but no other data structures, with statements such as DIMENSION, READ TAPE, REWIND, and BACKSPACE.

Table look-up was also solved early through hashing. The software pioneers of the first decade did not look beyond address computation because memories were so small that any structure that "wastes" space on pointers was considered a luxury. Memories containing of a few K words only restricted programmers to only the very simplest of data structures, and the limited class of problems addressed let them get away with it. The discipline of data structures had not yet been born.

2.2 Commercial data: Batch processing of dynamic sets, single key access

Commercial data processing led to the most prolific phase in the development of data structures, comprehensively presented in Knuth's pioneering books on "The Art of Computer Programming" [K 68, 73]. These applications brought an entirely different set of requirements for managing data typically organized according to a single 'primary key'. When updating an ordered master file with unordered transaction files, sorting and merging algorithms determine data access patterns. The emergence of disk drives extended the challenge of data structure design to secondary storage devices. Bridging the 'memory-speed gap' became the dominant practical problem. Central memory and disk both look like random access devices, but they differ in the order of magnitude of two key parameters:

	Memory	Disk	Ratio
Access time (seconds):	10^{-6}	$10^{-2} - 10^{-1}$	$10^{4} - 10^{5}$
Size of transfer unit (bits):	$10 - 10^{2}$	10^{4}	$10^{3} - 10^{2}$

In recent decades technology has reduced both time–parameters individually, but their ratio has remained a 'speed gap' of 4 orders of magnitude. This fact makes the number of disk accesses the most relevant performance parameter of data structures. Many data structures make effective use of central memory, but disk forces us to be more selective; we need data structures that avoid pointer chains that indiscriminately cross disk block boundaries. The game of designing data structures suitable for disk has two main rules: the easy one is to use a small amount of central memory effectively to describe the current allocation of data on disk in a way that facilitates rapid retrieval; the hard one is to ensure that the structure adapts gracefully to the ever-changing content of the file.

Index-sequential access methods (ISAM) order records according to a single key so that a small directory, preferably kept in central memory, ideally directs any point query to the correct data bucket where the corresponding record is stored, if it is present at all. But the task of maintaining this single-disk-access performance in a dynamic file, in the presence of insertions and deletions, is far from trivial. The first widely used idea splits storage into two areas, a primary and an overflow area. It suffers from several well-known defects: 1) Once the primary area has been allocated, it is not easily extended, even if there is space on disk, as an extension may force data to be moved between many buckets. 2) Insertions or deletions of real data are usually biased (non-uniformly distributed and dependent), so some primary buckets will become the heads of long overflow chains, and others may become sparsely populated. The former degrades access time, the latter storage utilization. Many studies have reached the conclusion that ISAM with overflow works effectively only as long as the amount of data in the overflow area is below about 20%. With the high rate of change typical of transaction processing, this bound is reached quickly, causing frequent reorganization of the entire file.

Balanced trees of any kind [e.g. AL 62, BM 72, and many others] provided a brilliant solution to the problem of 'maintaining large ordered indexes' without degradation: Frequent small rebalancing operations that work in logarithmic time eliminate the need for periodic reorganization of the entire file.

Trees based on comparative search derive their strength from the ease of modifying list structures in central memory. They have been so successful that we tend to apply and generalize them beyond their natural limitations. In addition to concerns about the suitability of comparative search trees for multikey access, discussed in the next section, these limitations include [Ni 81]:
1) The number of disk accesses grows with the height of the tree. The fan-out from a node, or from a page containing many nodes, varies significantly (from dozens to hundreds) depending on page size, page occupancy, and the space required to store a key. For files containing 10^6 records, the tree may well have more than 2 or 3 levels, making the goal of "instantaneous" retrieval impossible.
2) Concurrency. Every node in a tree is the sole entry point to the entire subtree rooted at that node, and thus a bottleneck for concurrent processes that pass through it, even if they access different physical storage units. Early papers (e.g. [BS 77, KL 80) showed that concurrent access to trees implemented as lists requires elaborate protocols to insure integrity of the data.

2.3 Transaction processing: Interactive multikey access to dynamic sets

Whereas single-key access may suffice for batch processing, transaction processing, as used in a reservations or banking system, calls for for multikey access (by name, date, location, ..). The simplest ideas were tried first. Inverted files try to salvage single-key structures by ordering data according to a 'primary key', and 'inverting' the resulting file with respect to all other keys, called 'secondary'. Whereas the primary directory is compact as in ISAM, the secondary directories are voluminous: Typically, each one has an entry for every record. Just updating the directories makes insertions and deletions time-consuming.

Comparative search trees enhanced ISAM by eliminating the need for overflow chains, so it was natural to generalize them to multikey access and improve on inverted files. This is easy enough, as first shown by k-d trees [Be 75]. But the resulting multi-key structures are neither as elegant nor as efficient as in the single-key case. The main hindrance is that no total order can be imposed on multidimensional space without destroying some proximity relationships. As a consequence, the simple rebalancing operations that work for single-key trees fail, and rebalancing algorithms must resort to more complicated and less efficient techniques, such as general dynamization [Wi 78, Ov 81].
Variations and improvements on multidimensional comparative search trees continue to appear [e.g. LS 89]. Their main virtue, acceptable worst case bounds, comes from the fact that they partition the actual data to be stored into (nearly) equal parts. The other side of this coin is that data is partitioned *regardless of where in space it is located.* Thus the resulting space partitions exhibit no regularity, in marked contrast to radix partitions that organize space into cells of predetermined size and location.

2.4 Knowledge representation: Associative recall in random nets

There is a class of applications where data is most naturally thought of as a graph, or network, with nodes corresponding to entities and arcs to relationships among these. Library catalogs in information retrieval, hypertexts with their many links, semantic nets in artificial intelligence are examples. The characteristic access pattern is 'browsing': A probe into the net followed by a walk to adjacent nodes. Typically, a node is not accessed because of any inherent characteristic, but because it is associated with (linked to) a node currently being visited. The requirements posed by this type of problem triggered the development of list processing techniques and list processing languages.

These graphs look arbitrary, and the access patterns look like random walks - neither exhibit any regular structure to be exploited. The general list structures designed for these applications have not evolved much since list processing was created, at least not when compared to the other data structures discussed. The resulting lack of sophisticated data structures for processing data collections linked as arbitrary graphs reminds us that efficient algorithms and data structures are always tailored to specific properties to be exploited. The next application shows such a special case.

2.5 Spatial data management: Proximity access to objects embedded in space

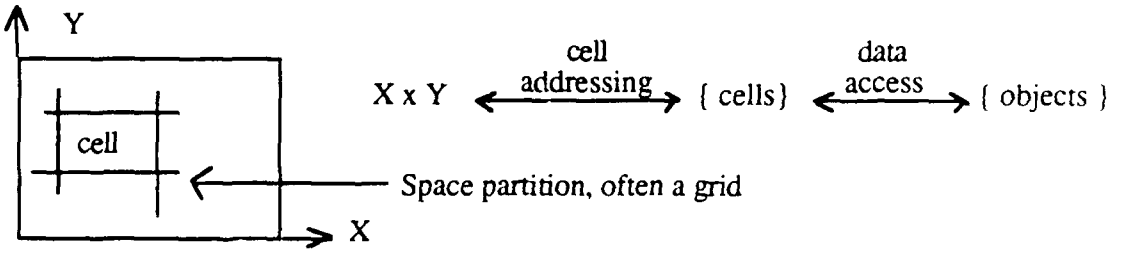
Three key characteristics set spatial data management apart from the four areas described above:
1) Data represents objects embedded in some d-dimensional Euclidian space $\Re^d$.
2) These objects are mostly accessed through their location in space, in response to a proximity query such as intersection or containment in some query region.
3) A typical spatial object has a significantly more complex structure than a 'record' in the other applications mentioned.
Although other applications share some of these characteristics to a small extent, in no other do they play a comparably important role. Let us highlight the contrast with the example of a collection of records, each with two attributes, 'social security number' and 'year of birth'.
1) Although it may be convenient to consider such a record to be a point in a 2-d attribute space, this is not a Euclidian space; the distance between two such points, for example, or even the distance between two SSNs, is unlikely to be meaningful.
2) Partial match and orthogonal range queries are common in data processing applications, but more complex query regions are rare. In contrast, arbitrarily complex query regions are common in geometric computation (e.g. intersection of objects, or ray tracing).
3) Although a record in commercial data processing may contain a lot of data, for search purposes it is just a point. A typical spatial object, on the other hand, is a polyhedron of arbitrary complexity, and we face the additional problem of representing it using predefined primitives, such as points, edges, triangles, tetrahedra.

3 Representation of space

This section describes the different ways space is represented by various approaches to spatial data structures. In each case, space is partitioned, but the regularity of the partition varies from one to another, as does the method of access.

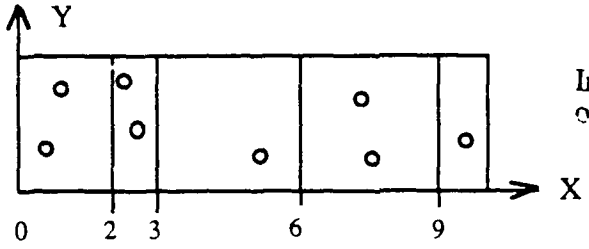
3.1 Space partitioning and cell addressing

The first point to be made about spatial data structures sounds so trite that it is often overlooked: *The main task of a spatial data structure is to organize and represent space.* The way empty space is organized and represented has important consequences for the representation and storage of objects, much as the choice of a coordinate system has important consequences for the manipulation of geometric formulas. A representation of space suitable for data storage and management has two main components:
1. A scheme for partitioning the entire space into cells.
2. A two-way mapping that relates regions of space to the cells that represent them.
The space partition serves as a framework or skeleton for organizing objects located in space, and leads to a characteristic two-step procedure for accessing objects: Given a region of space, first find the cells that intersect the region (cell addressing), then find the objects that inhabit those cells (data access).



Posing the issue of space representation in this generality makes some useful observations obvious:
- For efficient addressing, cells must have a simple shape, usually boxes.
- The simplest space partitions are orthogonal grids.
- We want to control the size of cells, most easily done by means of hierarchical grids.

These desiderata lead directly to radix trees, the archetypal hierarchical space partitions, discussed in the next section. But first, consider an inverted file as an example of a data structure that was not designed for spatial access. Like any data structure, it partitions space by the way it allocates data to buckets. If x is the primary key, and y a secondary key, space is cut up into slices orthogonal to the x-axis, thus revealing that it is really a 1-d data structure. Inverted files support queries with specified x-values efficiently, but not with y.



Inverted file (shown managing data buckets of capacity 2) partitions space into slices.

We now turn our attention to hierarchical space partitions, which come in two major classes: radix trees and comparative-search trees.

3.2 Radix trees in various dimensions: quadtrees, oct-trees, r^d-trees. Grid file

All general-purpose schemes for partitioning space into regions must permit refinement, and thus are naturally hierarchical. Radix trees, more exactly r^d–trees, apply to arbitrary dimension d and radix $r > 1$ and generate the most easily computed refinable space partitions. The quad tree (the case d=2, r=2) and oct-tree (d=3, r=2) are the most familiar radix trees. The picture below shows the space partition generated by a quad tree and three ways of addressing its cells.

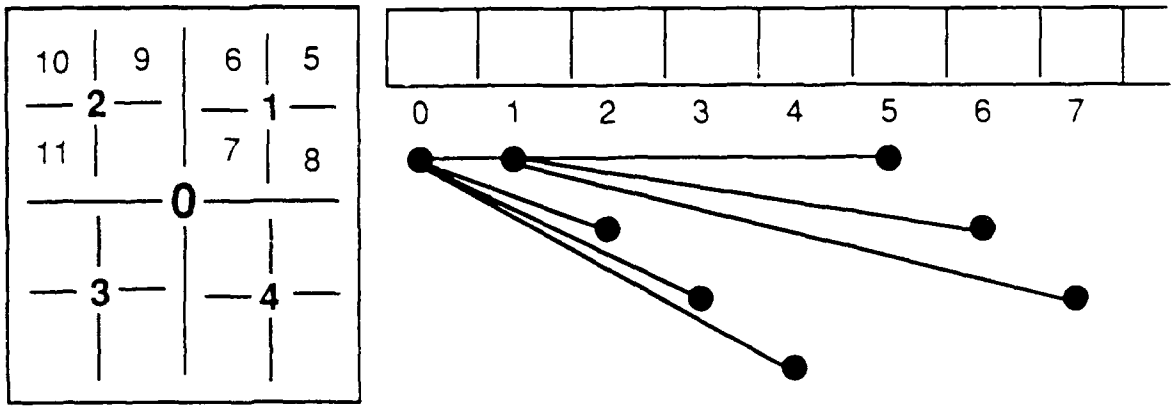
1) Breadth-first addressing leads to the simplest address computation based on formulas that make tree traversal very efficient: the node at address (index) i has children at addresses 4i+1, 4i+2, 4i+3, 4i+4 conversely, the node at address j has its parent at (j-1) div 4. The analogous breadth-first storage of a binary tree (d=1, r=2) is well known as the key idea that makes heap sort efficient and elegant.

2) Bit interleaving, or z-order, is another way to treat all dimensions (axes) symmetrically. The (linear) address of a cell at coordinates (x, y), for example, is obtained by writing x and y as binary integers x = x1 x0, y =y1 y0, and constructing the address a as the binary integer a = x1 y1 x0 y0.

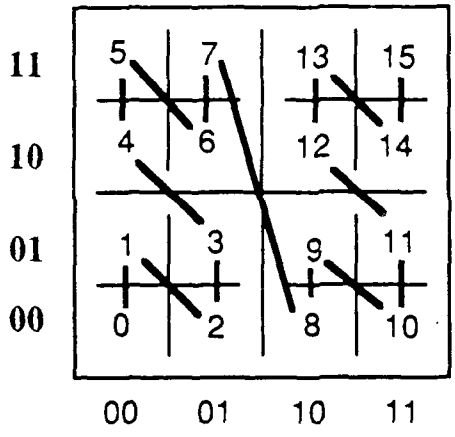
3) Path addressing assigns a string over the alphabet { 1, .., r } to each cell. The null string λ to the entire space, strings of length 1 to the quadrants at depth 1, strings of length 2 to the subquadrants at depth 2, etc. When these strings are ordered lexicographically we obtain breadth-first addressing.

The point of listing these addressing schemes is not to argue about minor advantages or disadvantages, but to show that r^d–trees partition space in such a regular manner that any reasonable way of numbering them leads to a simple and efficient computation that relates a cell identifier to the region of space occupied by the cell.

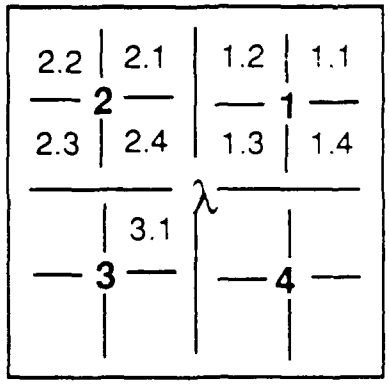
Breadth-first addressing: parent i <——> children 4i + { 1, 2, 3, 4 }



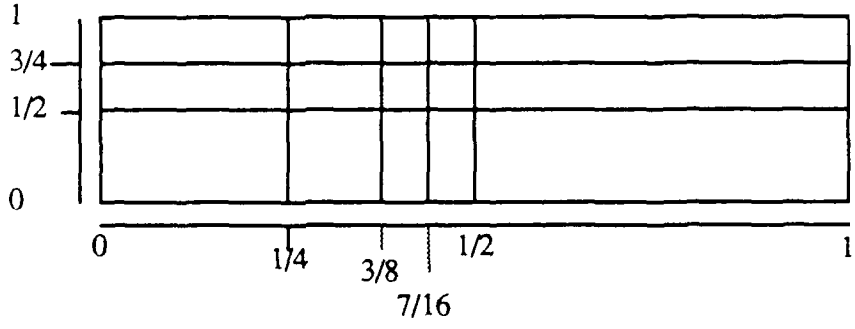
Bit interleaving



Path addressing



The grid file [NHS 84], [Hin 85] space partition is almost an r^d-tree partition: Boundaries are introduced at multiples of powers of a radix r, e.g. at 1/2, 1/4, 3/4 as in a quad tree. But each dimension is partitioned independently of the others, and a boundary cuts across the entire space. A grid file partition is compactly represented by 'linear scales', one per dimension, that make the computation of the intersection of a query region with the grid cells comparatively simple. The regularity of the grid partition, and its compact representation in scales, make it possible to develop a number of reasonable structures for managing such a dynamic grid, and several variations on the grid file have been introduced (e.g. [Fr 87])

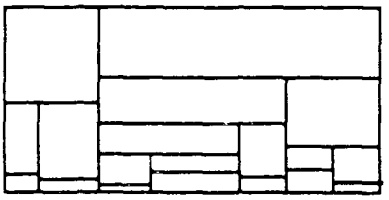


grid file space partition, and scales that define it

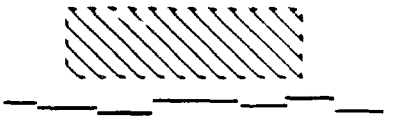
3.3 Space partitions induced by comparative search trees. Load balancing.

In contrast to the regularity of the r^d-tree space partition, partitions based on comparative search (such as k-d trees) are less regular. Drawing a boundary so as to balance the load among the left and right subtrees works much better for 1-d data than for multi-dimensional space. The result is shown in the figure below. Among the disadvantages:

- Crafty but complicated rebalancing techniques, called 'general dynamization' (e.g.[Wi 78], [Ov 81]), are necessary. These are less effective than the logarithmic rebalancing algorithms (such as those for height or weight-balanced trees) that work in totally ordered domains, but not w.r.t. the partial order natural in multidimensional spaces.
- Concurrent access requires elaborate protocols. The root of a tree represented as a list structure is a bottleneck through which all processes must travel as they access data (notice that this objection need not hold for radix trees, as these permit access by means of address computation). Any process with the potential to modify the tree structure may delay read-only processes in different leaves.
- A multitude of small boundaries (e.g. along the bottom of the figure) complicates the computation of queries as simple as orthogonal range queries.



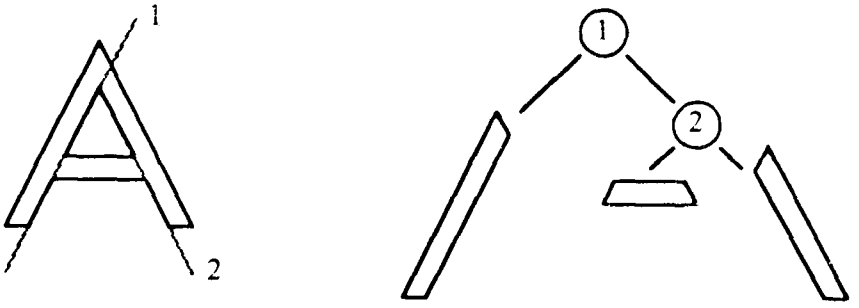
k-d tree space partition



3.4 Constrained space partitions: BSP trees, Octet

The space partitions presented so far have some structure that is independent of the objects that populate the space. Radix partitions, in particular, have a fixed 'skeleton', and the set of objects merely determines the degree of refinement. Space partitions for comparative search trees are more data-dependent, but there is some choice in placing boundaries. In some applications, such as computer graphics or finite element computations, we want space partitions that are determined by the objects to be represented. For these, the distinction between 'organizing space' and 'representing objects' gets blurred.

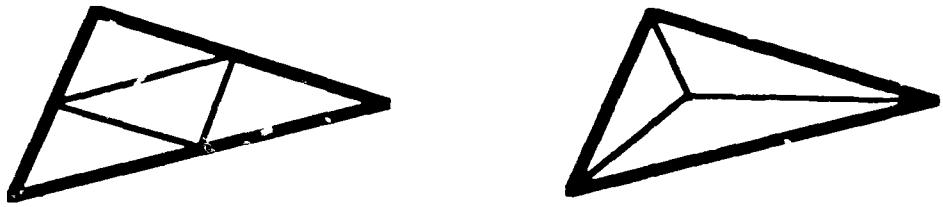
Binary space partition trees [FKN 80] were introduced to speed up image generation of objects in situations where the world model changes less frequently than the viewpoint of the observer. Consider an object consisting of many polygons in space. We select a suitable polygon (e.g. one whose plane f cuts only a few other polygons), and make it the root of a binary tree. Recursively, one of its subtrees partitions the half-space f^+ in front of f, the other the half-space f^- behind f. Polygons that stick out into both f^+ and f^- get cut into two. The following figure shows a 2-d example.



BSP tree partitions the letter A into quadrilaterals

We are working on Octet, a space partition scheme designed for mesh generation for 3-d finite element analysis. We typically have a static collection of objects, or, equivalently one large complex object. The space inside the object must be partitioned into cells that meet stringent conditions: (1) simple polyhedra, such as tetrahedra or hexahedra (distorted boxes), (2) with a good fit to the boundary of the objects, (3) that avoid excessively acute angles and excessive elongation, (4) that can be refined in such a way that condition (3) holds again for all subcells generated. Oct-trees do a great job on 1), 3) and 4), but not on 2) - even simple surfaces such as planes force the oct tree to its level of resolution unless they are orthogonal to an axis.

Octet partitions arbitrary 3-d regions of space hierarchically into tetrahedra and octahedra in such a way that aspect ratios of all cells (their elongation) is bounded. As an analogy, consider two ways of tessellating the plane into triangles, a good one and a bad one.



At left, the outer triangle is refined into 4 triangles of similar shape.
At right, into 3 increasingly elongated triangles.

The problem gets harder in 3-d space. It is clear how to refine a tetrahedron into 4 tetrahedra by introducing a new vertex in the middle, but that generates elongated tetrahedra that quickly become useless. The refinement analogous to the picture at left goes as follows. From each of the original vertices, cut off 4 tetrahedra that are all similar to the original: all linear extensions are halved, for a volume of 1/8 of the original. That leaves an octahedron at the core, whose volume is 1/2 of the volume of the original tetrahedron. In order to complete the recursion, we partition an octahedron into a smaller octahedron surrounded by 12 tetrahedra. This partition can be done in such a way that the aspect ratios of all the solids generated remain bounded, and can often be improved so that elongated solids generate more sphere-like subsolids.

4 Representing a useful class of objects

A representation for complex objects necessarily contains two distinct types of data:
- The primitive data items from which complex objects are built (points, line segments, surface patches, volume elements, ..).
- The relationships between these items (e.g., who touches whom, who belongs to whom).
Conventional object representations intermingle all this data, which serves a good purpose when drawing an object on the screen: Traversing the relationship structure one encounters the definition of all primitives, and can draw them as one goes along. But this intermingling can have a negative effect on retrieval performance, as I will argue in the next section.

These two types of data (primitive items and relationships) usually have rather distinct characteristics that suggest they should be treated separately. Many engineering applications use only a small number of different **types** of primitives from which they build complex structures with a rich, irregular network of relationships. For these cases we advocate a representation that separates primitives from relationships in the following manner.

Primitives. Each type of primitive is characterized by a number of parameters: a circle or sphere by its radius, an intervals by its length, a triangle by its shape and size. Thus an instance of each type is fully defined by giving the values of each of its parameters, and can be considered to be a point in its parameter space. Points are the simplest objects to store and retrieve, and many data structures do a creditable job in managing large collections of points. Geometric proximity queries start by identifying primitives that meet the query condition, and only access the relationship data as needed.

The **relationships** between the primitives are represented as a graph tied to the embedding space through the primitives.

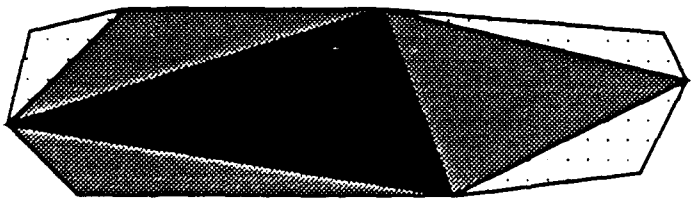
In geometry, it is useful to distinguish between two concepts that are often confused: An 'object' (e.g. a triangle) and an 'object embedded in space' (at a specific location). Thus we separate the position and orientation of an object from those of its properties that are invariant under translation and rotation. This distinction may be unnecessary if objects never move, but we are concerned with the increasing number of interactive applications of geometric computation where the data configuration is highly dynamic. All the representations we discuss allow us to make a clean distinction except one: In the technique that we call 'mark inhabited space', an object is represented, or at least approximated, by marking all the space cells it occupies. This is the traditional use of quad trees in image processing, for example, which supports only the concept 'object embedded in space' but lacks the notion of a 'generic object' independent of its location. Thus we do not consider this a true *object representation* technique. We discuss it in 5.1 as a scheme for *embedding* objects in space.

4.1 Geometric modeling: CSG, boundary and sweep representations

The great variety of techniques for representing spatial objects developed over the decades attest to the fact that there is no single best representation. Wire-frame and surface patch models, the early workhorses of computer graphics, were the precursors of the boundary representation (BR) technique most widely used today in solid modeling. BR is a general-purpose technique for handling any type of spatial objects and operations. Constructive solid geometry (CSG) and sweep representations are particularly suitable for relatively simple objects. We assume the reader is familiar with these solid modeling techniques. [Man 88] and [Hof 89] are recent survey books. When comparing these solid modeling techniques with other representations it is worth remarking that they were designed primarily for processing in central memory, not for their efficiency when retrieving objects from disk in response to geometric queries.

4.2 Hierarchical approximations

Computational geometry has developed some object representations that are highly effective in particular circumstances. Hierarchical representations approximate a convex object by piling layers of simple primitives on top of each other, as the following example illustrates. We inscribe a triangle as a level-0 approximation to a given polygon. This triangle is surrounded by a layer of level-1 triangles that are also inscribed in the target polygon. The approximation improves with each additional layer. The structure of these approximations is a tree, and if care has been taken to grow a balanced tree, some operations on an target object of size n can be done in time O(log n).



Hierarchical approximation of a convex 9-gon as a 3-level tree of triangles. The root is in black, its children in dark grey, grandchildren in light grey.

4.3 Object specification by means of parameters

In some applications, for example VLSI design, the data consists of a large number of very simple primitives, such as aligned rectangles in the plane (i.e. with sides parallel to the axes). In many more engineering applications, for example technical drawing, all objects are constructed from relatively few types of simple primitives, such as points, line segments, arcs, triangles, rectangles, or the 3-d counterparts of such typical primitives. Under these circumstances it is natural and often efficient to split the representation of a complex object into two parts: A graph or network that specifies the relationships among the instances of the primitives, and, for each type of primitive, an unstructured set of all the instances of the primitives.

Each instance of a primitive is given by as many parameters as are necessary to specify it within its type. Continuing the example of the class of aligned rectangles in the plane, we observe that each rectangle is determined by its center (cx, cy) and the length of each side, dx and dy. We consider the *size parameters* dx and dy to represent the object, and the *location parameters* cx and cy to specify the embedding of the object in the space. The representation of simple objects in terms of parameters becomes interesting when we consider how the metric in the object space gets transformed into parameter space - we continue this topic in section 5.3.

5 Schemes for embedding objects in space

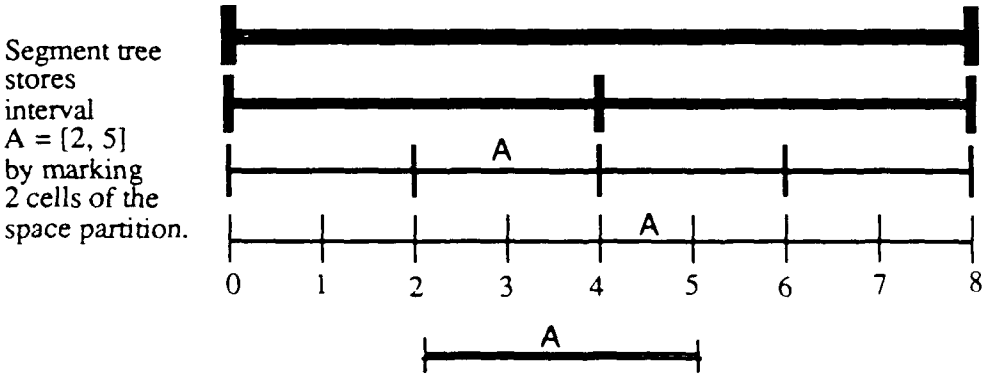
The question of how objects are represented can be asked, and partly answered, independently of how space is organized. But obviously, representation of space and of objects must be compatible, and their interrelationship is **the** key issue in designing spatial data structures. The problem is to ensure that the relationship between 1) regions of space, and 2) objects that inhabit these regions, can be computed efficiently **in both directions:** S → O, space-to-objects; and O → S, objects-to-space.

This section describes three techniques for embedding objects in space, ranging from a straightforward occupancy description to an embedding in a parameter space of higher dimensionality. It appears that increased abstraction provides increased power for the kinds of searches required of spatial data.

5.1 Mark inhabited space

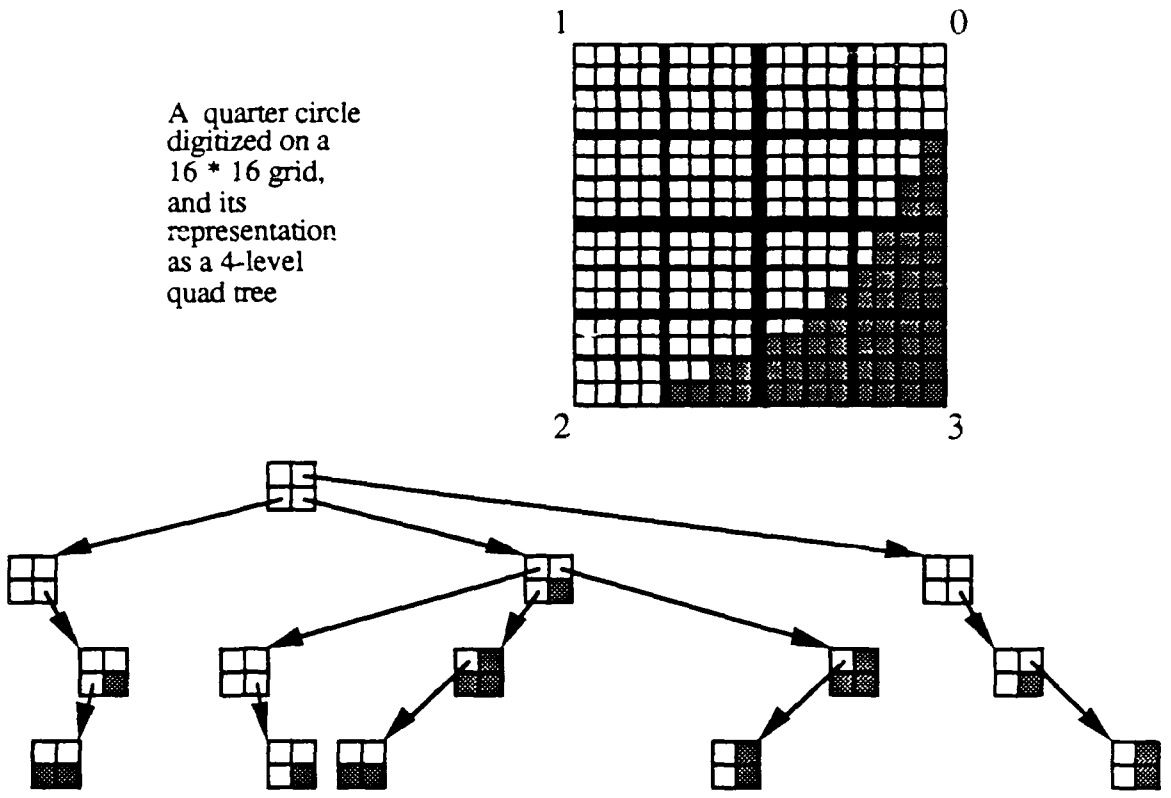
The most straightforward embedding is based on the idea that each object leaves its mark (name, identifier, pointer to itself) in every space cell that it occupies. This is the technique traditionally used with radix trees, illustrated by the segment tree and quad tree below. Some of its properties include:
- Potentially high redundancy; an object may require a lot of pointers to itself.
- S → O: as efficient as is practically possible;
- O → S: not as efficient as one might hope for: Many an object (e.g. a rectangle) has a simple description, but a radix tree forces us to break up the region it occupies.

The picture below shows the 1-d space [0, 8) hierarchically partitioned into cells by a binary tree (r = 2) of depth t = 3 One interval at level 0, 2 at level 1, 4 at level 2, $8 = 2^t$ at level t = 3. This scheme is called the 'buddy or twin system' in systems programming, 'segment tree' in computational geometry, and, for various radices r, is the basis for most systems of measurement.



The following picture of a digitized quadrant of a circle stored in a quad tree (d = 2, r = 2) shows not only the underlying space partitions, but also the correspondence between space cells and nodes in the tree. When used in image processing, the tree typically serves both as a directory and as actual storage. In other applications the tree is just a directory, with a leaf of the tree holding not just a black/white bit, but a pointer to all the data associated with the corresponding space cell. Notice that this technique has no way of capturing the concept of a quarter circle independent of its location - if the object is moved ever so slightly, its representation changes completely.

A quarter circle digitized on a 16 * 16 grid, and its representation as a 4-level quad tree



5.2 Anchors: representative points or vectors

Every object has some prominent point, such as the center of gravity, or the lower left corner, that can be used to anchor an object in space. Many practical objects also have a prominent axis, such as a symmetry axis, that is conveniently used to orient the object. Such a representative point or vector is an **anchor**: if all the parameters of the anchor are specified, the object is located (anchored) in space.

CSG, boundary and sweep representations often use coordinates relative to an anchor. Relative coordinates have the great advantage that common transformations, translations and rotations,become efficient O(1) operations: transforming the anchor transforms the whole object.

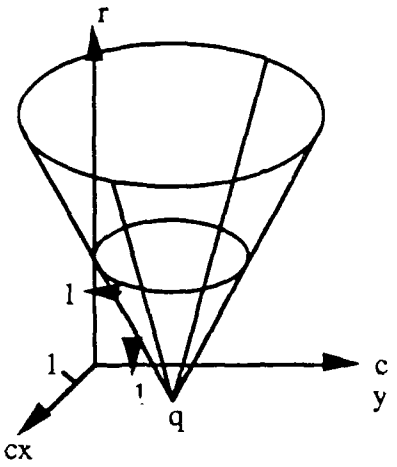
Both functions S → O (given a region, perhaps a cell, retrieve the objects in it) and O → S (given an object, what cells does it intersect) can be efficiently implemented if the objects satisfy constraints, as is often the case in practice. If we know a bound r on the size of all objects, for example, the sphere of radius r centered at the anchoring point becomes a container that often eliminates the need to examine objects in detail. If there are only a few distinct types of objects, anchoring becomes similar to the parameter specification approach of 5.3

5.3 Transformation to parameter space

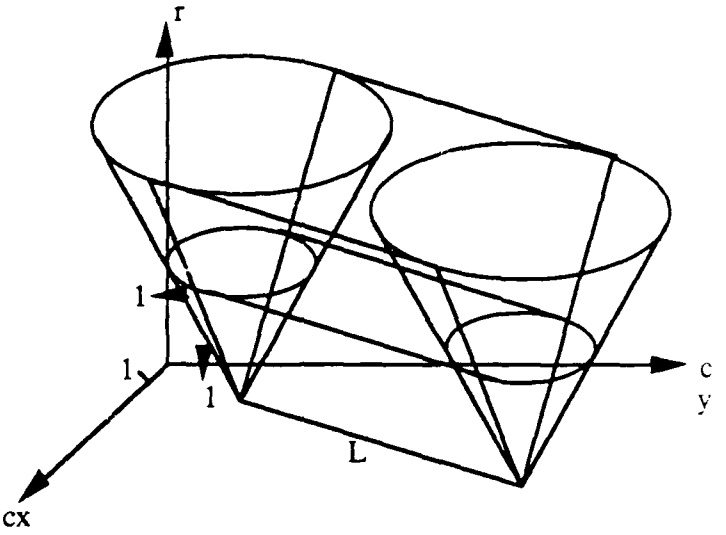
Given a parameter space assigned to a class of objects as in section 4.3, we construct the cartesian product with a space of *location parameters* to obtain a higher-dimensional space that tells us everything about a collection of objects of this type located in space [NH 85]. As an example, consider a collection of circles in the plane. Each instance of a circle is specified by its radius, the size parameter r, and by its center, given by two location parameters cx and cy. Thus the collection of circles to be stored is transformed into a set of points in the parameter space with axes cx, cy, r shown in the two figures below.

Under this transformation, common proximity queries in object space turn into region queries in parameter space. Continuing with the example of the circles, it is straightforward to verify that the point query 'retrieve all circles that cover the point q' gets transformed into the region query 'retrieve all points

in parameter space that lie in the search cone shown in the figure at left'. A region query in the object space of circles, such as 'retrieve all circles that intersect (or contain) the line L' is a union of point queries and gets transformed into the region query which is the union of search cones, namely 'retrieve all points in parameter space that lie in the search region shown in the figure at right'.



Search cone for a point query in the class of circles in the plane.



Search region for an intersection query with a line L

Transformation to parameter space reduces object retrieval to point retrieval in a parameter space that is typically of higher dimensionality than the original object space. And it generates region queries of a large variety of shapes. The grid file was designed to evaluate complex region queries with a minimum of disk accesses. The data buckets that may contain points in the search region are identified on the basis of a small amount of directory information (the 'scales') that is kept in central memory. Thus a search region of complex shape affects the computation time, but not the number of disk accesses - the latter are determined primarily by the size of the answer.

6 Support for geometric transformations

Previous sections addressed the basic questions of how to represent space and objects, and how to embed objects in space. This and the following two sections address three issues of primary importance for the efficiency of spatial data structures: Geometric transformations, proximity searches, and access paths. While not an exhaustive list of geometric operations, these three aspects provide important insights into how well a spatial data structure performs.

Geometry is the study of object-properties that remain invariant under some group of geometric transformations. Thus a general purpose spatial data structure must be judged according to the efficiency with which the most common transformations can be performed, in particular linear transformations such as translation, rotation, and scaling (stretching or shrinking).

The efficiency of the two most frequently used transformations, translation and rotation, depends primarily on the scheme chosen for embedding objects in space. All embeddings based on the principle 'mark inhabited space' fare poorly in this respect. But the other embeddings we discussed all have the potential of transforming an object of arbitrary complexity as an O(1) operation. The common geometric models used in CAD, by using relative coordinates, make it possible to embed objects using the anchoring technique. Parameter space representations also make it possible to separate the six location and orientation parameters from the size parameters.

The efficiency of scaling depends primarily on the object-representation scheme, but in contrast to translation and rotation, few schemes can do better than O(n), the complexity of the object.

7 Proximity search: Simplifying complex objects to access disk sparingly

The most basic query a spatial data structure must answer efficiently is of the type 'retrieve the object(s) at or near point p = (x, y, ..)'. This *point-proximity search* is the prototype on which more complex proximity queries are based (region queries, intersection, containment, etc.).

The main idea that serves to speed up proximity search is to make the *processing effort independent of the complexity of the objects involved.* Although this ideal cannot be achieved in general, it can often be approximated fairly well, in at least two ways:

1) Certain objects of complexity n can be represented exactly in terms of a structure that permits proximity search in time o(n), mostly in logarithmic time O (log n), or even in constant time O(1). Example: The hierarchical representation of a convex polygon or polyhedron (see section 4.2) serves to answer the point-in-object query in time O(log n). Although this search time does depend on the complexity n of the object, for practical values of n the difference between logarithmic time and constant time may be insignificant.

2) Approximation of a complex object by a simpler one can provide fast answers for most queries. The two most prominent kinds of approximations are containers and kernels. Both serve to replace costly proximity searches by cheaper ones. Containers are used when we expect most searches to fail, kernels when we expect them to succeed.
- By enclosing the object in a simple container such as a bounding box or a circumscribed sphere we achieve the saving that when the low-cost search for the container fails, we know that the expensive search for the actual object must also fail. Only when the container search succeeds we must follow up with a search for the object.
- By inscribing a simple kernel, such as a sphere or tetrahedron, inside an object we achieve the analogous effect. Successful kernel searches give a definite answer, only failed kernel searches must be followed by the costlier object search.

Different classes of objects require different types of containers or kernels if the approximation is to be effective. For example, aligned objects with sides parallel to the coordinate axes call for box-shaped approximations, whereas rounded object are better approximated by spheres or ellipsoids. A data structure that permits rapid answers to the point-proximity query for standard containers or kernels such as aligned boxes, tetrahedra, and spheres provides a basis for the efficient implementation of proximity search for a more general class of objects also.

Among all the geometric operations performed on a collection of objects stored on disk, proximity search is the most critical in terms of its potential for saving or wasting disk accesses. Disk accesses are not the bottleneck of most other operations: These rely on proximity search to identify the objects they must process, read them off disk, then spend considerable time processing this data. Proximity search is the main filter that takes a quick look at many objects, in the hope of discarding most of them. Simple approximations to complex objects may make it possible to keep the containers (only) of most objects in central memory, and be highly selective in accessing entire objects.

8 Support for the data access patterns of typical geometric algorithms

Having discussed separately the two most important special cases of object processing, namely transformation and proximity search, we now attempt to characterize the requirements imposed by arbitrary algorithms that may be used to process objects. The majority of the practical geometric algorithms we are aware of fall naturally into three classes:
- Sweeping the plane with a line, or sweeping 3-d space with a plane.
- Boundary traversal
- Recursive data partitioning in divide-and-conquer algorithms.

Each of these types of algorithms generates distinct data access patterns.

Sweep algorithms ask for the data sorted in lexicographic order: by increasing x, for equal x by increasing y, etc. This is by far the easiest data access pattern to implement efficiently. Indeed, the initial step of sorting the data dominates the work in practically all plane-sweep algorithms: Most of them run in time O(n log n), the time complexity of sorting, and some of them run in linear time O(n) on data that is presorted. Data access in sweep algorithms exhibits locality both in object space and in address space: In object space, because we move from one event (typically a point) to the next event to the right; in address space, because lexicographically sorted data is easily mapped *monotonically* into a linear address space.

Boundary traversal algorithms start at some point of a boundary line or surface and march from a vertex to an adjacent vertex selected according to local conditions. Computing the intersection of two surfaces serves as an example: Once any intersection point has been found, one follows the intersection line. Data access in boundary traversal satisfies a locality principle in object space, but usually not in address space: when data is allocated, we have no way of knowing which one among several neighboring vertices will be visited next. Thus most geometric neighbors reside far apart in address space. This is no problem as long as all the data resides in memory, but is likely to cause many page faults when data is processed off disk.

Recursive data partitioning typically generates the most irregular data access patterns. A recursive computation may hop around its data at 'random', but even when it can be sequenced to exhibit locality in object space, not much locality in address space is gained. This is simply because of the ever-present problem that proximity in multi-dimensional space gets lost under a mapping to 1-d address space.

A systematic experimental investigation of how well different spatial data strucures support these access patterns would be useful, but we are unaware of any.

9 Implementation: Reconciling conceptual simplicity and efficiency

Geometric computation and data bases, the two main forces that affect the development of spatial data structures, evolved independently, emphasizing goals that are often incompatible. Computational geometry focused on sophisticated algorithms tailored to a particular problem, using intricate data structures to obtain asymptotic optimality. Data bases, on the other hand, emphasized very general structures that can model anything by breaking complex structures into constituent pieces and using the decomposition relationships as access paths, with a resulting loss of direct access.

Neither of these two extreme points serve as a good model for implementations. Practical software requires a *careful balance* between conceptual simplicity, which leads to understandable programs, and sophisticated algorithms and data structures, which lead to efficiency. If a data structure is implemented in a stand-alone test program, this point may not be of great importance. But when it is used as a component in a complex data management system, this point can hardly be overemphasized. The data structures mentioned have proven their value as systems components, and we urge designers of new ones to attach as great an importance to this aspect of implementability as to any of the other criteria.

Acknowledgement. I am grateful to Klaus Hinrichs, Hans Hinterberger, Peter Schorn, and Don Stanat for helpful comments. This work was supported in part by the National Science Foundation under grant DCR 8518796.

References

[AL 62] G. M. Adelson-Velskii, Y. M. Landis: An algorithm for the organization of information (in Russian), Dokl. Akad. Nauk SSSR, Vol 146, 263-266, 1962,

[BM 72] R. Bayer, E. M. McCreight: Organization and maintenance of large ordered indexes, Acta Informatica, Vol 1, 173-189, 1972

[BS 77] R. Bayer, M. Schkolnick: Concurrency of operations on B-trees, Acta Informatica, Vol 9, 1-21, 1977 .

[Be 75] J. L. Bentley: Multidimensional binary search trees used for associative searching, Comm. ACM, Vol 18, No 9, 509-517, Sep 1975.

[EL 80] H. Edelsbrunner, J. van Leeuwen: Multidimensional algorithms and data structures (bibliography), Bulletin of the EATCS, 1980.

[Fr 87] M. W. Freeston: The Bang file: a new kind of grid file, Proc. ACM SIGMOD Conf. 1987.

[FKN 80] H. Fuchs, Z.M. Kedem, B.F. Naylor: On visible surface generation by priority tree structures, Computer Graphics (Proc. SIGGRAPH '80), Vol 14, 3, 123-133, 1980.

[Gun 88] O. Gunther: Efficient structures for geometric data management, Lecture Notes in CS, 337, Springer 1988

[Gut 84] A. Guttman: R-trees: a dynamic index structure for spatial searching, Proc. ACM SIGMOD Conf. on Management of Data, 47-57, 1984.

[HSW 89] A. Henrich, H-W. Six, P. Widmayer: The LSD tree: spatial access to multidimensional point- and non-point-objects, Proc. VLDB, Amsterdam, 1989.

[Hin 85] K. Hinrichs: Implementation of the grid file: design concepts and experience, BIT 25 (1985), 569 - 592.

[Hof 89] C. M. Hoffmann: Geometric and solid modeling, Morgan Kaufman Publ., 1989.

[K 68,73] D. E. Knuth: The Art of Computer Programming, Addison-Wesley.
Vol 1 Fundamental Algorithms, 1968; Vol 3 Sorting and Searching, 1973.

[KL 80] H. T. Kung, P. L. Lehman: Concurrent manipulation of binary search trees, ACM Trans. Database Sys., Vol 5, No 3, 354-382, Sep 1980.

[LS 89] D. Lomet, B. Salzberg: The hB-tree: A robust multiattribute search structure, Proc. 5-th Int. Conf. on Data Engineering, Feb 1989.

[Man 88] M. Mantyla: An introduction to solid modeling, Computer Science Press, Rockville, MD, 1988.

[Me 84] K. Mehlhorn: Data structures and algorithms, Vol 3, Multi-dimensional search and computational geometry, Springer, 1984.

[Ni 81] J. Nievergelt: Trees as data and file structures. In CAAP '81, Proc. 6th Coll on Trees in Algebra and Progamming, (E. Astesiano and C. Bohm, eds.), Lecture Notes in Computer Science, 112, 35-45, Springer 1981.

[NHS 84] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The grid file: an adaptable, symmetric multikey file structure, ACM Trans. on Database Systems 9, 1, 38 - 71, 1984.

[NH 85] J. Nievergelt, K. Hinrichs: Storage and access structures for geometric data bases, Proc. Kyoto 85 Intern. Conf on Foundations of Data Structures (eds. Ghosh et al.), 441-455, Plenum Press, NY 1987.

[Ov 81] M. H. Overmars: Dynamization of order decomposable set problems, J. Algorithms, Vol 2, 245-260, 1981.

[PS 85] F. Preparata, M. Shamos: Computational Geometry, Springer-Verlag, 1985.

[Sa 89] H. Samet: The design and analysis of spatial data structures, and Applications of spatial data structures, Addison Wesley, 1989

[SK 88] B. Seeger, H-P. Kriegel: Techniques for design and implementation of efficient spatial access methods, 360-371, Proc. 14-th VLDB, 1988.

[SRF 87] T. Sellis, N. Roussopoulos, C. Faloutsos: The R^+-tree: A dynamic index for multidimensional objects, 507-518, Proc. VLDB, 1987.

[Wi 78] D. E. Willard: Balanced forests of h-d trees as a dynamic data structure, Harvard Report, 1978.